

A Program Dependence Model for Concurrent Logic Programs and Its Applications

Jianjun Zhao
Department of Computer Science
and Engineering
Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Higashi-ku
Fukuoka 811-02, Japan
zhao@cs.fit.ac.jp

Jingde Cheng
Department of Information and
Computer Sciences
Faculty of Engineering
Saitama University
255 Shimo-Okubo, Urawa 338-8570, Japan
cheng@ics.saitama-u.ac.jp

Kazuo Ushijima
Institute of Systems & Information Technologies/KYUSHU
Fukuoka SRP Center Building 7F
2-1-22 Momochihama, Sawara-ku, Fukuoka 814-0001, Japan
ushijima@isit.or.jp

Abstract

In this paper we propose a program dependence model for concurrent logic programs. We present three types of primary program dependences named the sharing dependence, communication dependence, and unification dependence between arguments in a concurrent logic program. We formally define these primary program dependences based on mode information. We further present a dependence-based representation named the argument dependence net (ADN), which explicitly represents all primary program dependences between arguments in a concurrent logic program. We also discuss some applications of the ADN for developing software engineering tools of concurrent logic programs. Finally, we briefly introduce a program analysis system called CLPKIDS for concurrent logic programs, which is a prototype implementation of the techniques introduced in this paper.

1 Introduction

Program analysis tools are one of basic components for any programming environments. Proper program analysis tools can help programmers develop high quality softwares and save their much time on analyzing and maintaining softwares. However, on the other hand, if lack of program analysis tools, programmers may probably produce low quality softwares and have to spend much time on maintaining such softwares. It has been pointed out that logic programming languages, although have a strong theoretical basis for program analysis than many of other kinds of programming languages, research on logic programming environments is still not satisfied in comparison with the needs [7].

Program dependences are dependence relationships holding between statements in a program that are determined by control flows and data flows in the program. Many compiler optimizations depend on program dependence analysis, which is typically represented in the form of a *program dependence graph* (PDG) [8]. The PDG, although originally proposed for compiler optimizations, has been applied to various software engi-

neering activities including program slicing, debugging, testing, maintenance, and complexity measurements for imperative programs [4, 12, 14, 22]. For example, program slicing, a decomposition technique that extracts from program statements related to a particular computation, is greatly benefit from it where slicing problem can be reduced to a vertex reachability problem [12] that is much simpler than its original algorithm [19].

Program dependence analysis has also been studied in logic programming community. For sequential logic programs, program dependence information has been used for compile-time optimization of backtracking and parallelization [3], prediction of the run-time goal independence which can eliminate costly run-time checks in AND-parallel execution, and computation of the space complexity in the cost analysis [6]. For concurrent logic programs, program dependence information has been used for sequentialization of concurrent logic programs for more efficient multiprocessor executions [10, 11], and direction of the execution of AKL programs dynamically to improve the parallel execution [1]. However, most of work on program dependence analysis of logic programs aims at generating efficient codes, rather than developing program analysis tools to help programmers understand, debug, test, and maintain programs from a software engineering viewpoint. To our knowledge, little research has investigated the problem of using program dependence information to develop program analysis tools for logic programs. Gyimothy and Paakki [9] used program dependence graphs to slice sequential logic programs, especially Prolog programs. Schoenig and Ducasse [16] also use program dependence information to compute backward slices of Prolog programs. Pereira [13] used the term dependence information to dynamically improve the efficiency of algorithmic debugging of sequential logic programs. Zhao *et al.* [20] used a dependence graph-based approach to slicing concurrent logic programs.

Since program dependence analysis and a dependence-based representation are very useful in developing software engineering tools for imperative programs, using

program dependence information and a dependence-based representation to develop program analysis tools for concurrent logic program is reasonable. However, the existing definitions of program dependences in concurrent logic programs only considered the dependence relationships between literals (atoms) [1, 10, 11], and therefore have some problems when they are applied to develop software engineering tools:

First, the definition of program dependences is too ambiguous for understanding the total behavior of a concurrent logic program.

To perform program dependence analysis on concurrent logic program, two main features of concurrent logic program must be considered, that is: *interprocess synchronization* and *communication* consisting in suspending matching and shared variables between two literals in a clause, and *don't care non-determinism* consisting in guarded clauses with a commit operator. Existing techniques on program dependence analysis of concurrent logic programs [1, 10, 11] can handle neither of these features well. This is probably because that the aim of such research is originally motivated from problems such as how to efficiently implement and optimize codes of concurrent logic programs. As a result, it is unnecessary to distinguish different program dependence types. However, when we intend to develop program analysis tools by using program dependence information, proper distinction of program dependence types can often help us understand programs better. For example, highlighting program dependences concerning interprocess communications may greatly help us understand the concurrency features of a concurrent logic program, and therefore can aid us to detect some communication errors such as deadlocks and to measure the complexity of concurrency of a concurrent logic program.

Second, the definition of program dependences is too coarse for developing program analysis tools from a software engineering viewpoint.

The definition of program dependences for concurrent logic programs originally arose from the question of how to sequentialize a concurrent logic program into threads in order to efficient implementation of a concurrent logic programming language [10, 11]. For this purpose, program dependences between arguments are auxiliary, and only program dependences between literals are left. As a result, one can not determine which argument at one point of interest might affect another argument at another point of interest. Moreover, recent work has shown that program dependence information between arguments should be a basic requirement for slicing and debugging logic programs [13, 9, 16] since more precise slices could be computed by such information.

In order to solve these problems, we propose a new program dependence model for concurrent logic programs. We present three types of primary program dependences named the *sharing dependence*, *communication dependence*, and *unification dependence* between arguments in a concurrent logic program. Sharing dependences are used to represent the data flows in a single clause due to shared variables. Communication dependences reflect the data flows in a single clause due to interprocess communications. Unification dependences reflects the data flows between different clauses in a concurrent logic program. We formally define these primary program dependences based on mode information. We further present a dependence-based representation named the *argument dependence net* (ADN), which ex-

plicitly represents all primary program dependences between arguments in a concurrent logic program. We also discuss some applications of the ADN for developing software engineering tools of concurrent logic programs. Finally, we briefly introduce a program analysis system called CLPKIDS for concurrent logic programs, which is a prototype implementation of the techniques introduced in this paper.

The rest of the paper is organized as follows. Section 2 gives the preliminaries which are used in the following sections. Section 3 defines three types of primary program dependences in a concurrent logic program and presents a program representation named the argument dependence net for concurrent logic programs. Section 4 describes some applications of the argument dependence net. Section 5 briefly discusses some implementation issues of our program analysis system that supports the development of software engineering tools for concurrent logic programs, and conclusions are given in Section 6.

2 Preliminaries

We assume that readers are familiar with the basic concepts of logic programs, and in this paper, we will use KL1 [18], a concurrent logic programming language based on Guarded Horn Clauses (GHC), as our target language. This language illustrates the basic mechanisms of concurrent logic programming.

A *term* is a variable, a constant, or a compound term $f(t_1, \dots, t_n)$ where f is an n -ary function symbol and the t_i are terms, $1 \leq i \leq n$. An *atom* is of the form $p(t_1, \dots, t_n)$, where p is an n -ary *predicate* symbol and the t_i are terms called *arguments*, $1 \leq i \leq n$. A *literal* is either an *atom* or the negation of an atom. The number of arguments of a literal is called its *arity*.

A *guarded clause* is a formula of the form: $H :- G_1, G_2, \dots, G_n | B_1, B_2, \dots, B_m$ ($m, n \geq 0$), where H, B_1, B_2, \dots, B_m are literals, and G_1, G_2, \dots, G_n are guard test predicates. H is called the head of the clause, G_1, G_2, \dots, G_n are called the *guard* of the clause, and B_1, B_2, \dots, B_m are called the *body* of the clause, respectively. “ $:-$ ”, read *if*, denotes implication, and “ $|$ ” is called the *commit operator*. If the guard is empty the clause is written as: $H :- B_1, B_2, \dots, B_m$. and the commit operator is omitted. If the body is empty and the guard is not empty, the clause is written as: $H :- G_1, G_2, \dots, G_n | \text{true}$. If both the guard and the body are empty the clause is called an *unit clause*, and is written simply as: H . A clause whose body includes exactly one goal is called an *iterative clause*. A clause with only negative literals is referred to as a *goal*. A *procedure* is a set of clauses each of which has the same predicate name and arity. A KL1 *program* is a finite set of guarded clauses.

Figure 1 shows a sample KL1 program called STACK which implements a stack function. The stream of the first argument of procedure *stack* receives the list of the stack commands, and the stream of the second argument outputs the results. The stack is maintained in the stream of the third argument. The possible input commands are: *push(D)* pushes the value of D into the top of the stack; *pop* gets the value from the top of the stack and outputs it; *pop(N)* gets N values from the top of the stack and outputs them as a list with the length N; *reverse(N)* gets N values from the top of the stack and outputs them as a reverse list with the length N.

In order to formally define primary program dependences between arguments in a concurrent logic pro-

```

C1:  stack([push(D) | I], O, S) :-
      stack(I, O, [D | S]).

C2:  stack([pop | I], O, S) :-
      O = [A | NO],
      pop(A, S, NS),
      stack(I, NO, NS).

C3:  stack([pop(N) | I], O, S) :-
      O = [L | NO],
      pop(N, L, S, NS),
      stack(I, NO, NS).

C4:  stack([reverse(N) | I], O, S) :-
      O = [L | NO],
      rev:rev(R, L),
      pop(N, R, S, NS),
      stack(I, NO, NS).

C5:  stack([], O, _) :-
      O = [].

C6:  pop(A, [X | S], NS) :-
      A = answer(X),
      NS = S.

C7:  pop(A, [], NS) :-
      A = empty,
      NS = [].

C8:  pop(N, L, S, OS) :-
      N > 0 |
      L = [X | NL],
      pop(X, S, NS),
      NN = N - 1,
      pop(NN, NL, NS, OS).

C9:  pop(0, L, S, OS) :-
      L = [],
      OS = S.

```

Figure 1: A sample KL1 program.

gram, we distinguish each argument, literal, and clause of the program. We assume that the clauses of a concurrent logic program are denoted by C_1, C_2, \dots, C_n . The literals (including the guard test predicate) of a clause are numbered from left to right, such that the head is numbered by 0, the guard test predicate or the first body literal (if the guard is empty) is numbered by 1, and so on. The arguments of a literal are numbered from left to right by 1, 2, ...

To refer to an argument in a concurrent logic program, we use a method proposed by Boye *et al.* [2] which gives each argument of the program a unique label. If C_i is a clause, the position of the k^{th} argument of the j^{th} literal is uniquely defined in the program by the tuple (C_i, j, p, k) such that p is the predicate name of the j^{th} literal of C_i . Such a tuple is called an *argument position*.

3 Program Dependences in Concurrent Logic Programs and Their Representation

Program dependences are dependence relationships holding between program elements in a program that are determined by the control flows and data flows in

the program. Informally, if the computation of a program element directly or indirectly affects the computation of another program element, there might exist some program dependence between the elements.

Although program dependences are useful for compiler optimizations and development of software engineering tools for imperative programs, most of the work on program dependence analysis for logic programs still aims at generating efficient codes, rather than developing software engineering tools.

In this section, we propose a new program dependence model for concurrent logic programs. We present three types of primary program dependences named *sharing dependence*, *communication dependence*, and *unification dependence* between arguments in a concurrent logic program and a dependence-based representation named the *argument dependence net* (ADN), which explicitly represents the three types of primary program dependences in the program. As we will show in Section 4, an ADN can be used as an underlying model to develop program analysis tools for concurrent logic programs.

3.1 Primary Program Dependences

In a concurrent logic program, data can be transferred in two ways: either within a clause between two arguments sharing a common variable, or from one clause to another via unification. If we know the mode information of arguments in the program, we can further determine the direction that data is transferred, it means that we can determine data flows in the program. To represent such information in a concurrent logic program, we introduce two types of primary program dependences named *sharing dependence* which reflects data flows in a single clause due to sharing variables, and *communication dependence* which reflects data flows in a single clause due to interprocess communications. Moreover, according to the direction which data is transferred and the mode information of the arguments in a clause, we can further divide the sharing dependences into two categories: *backward-sharing dependence* which reflects the data-flow in a single clause from an argument of a head literal to an argument of a guard test predicate or a body literal, and *forward-sharing dependence* which reflects the data-flow in a single clause from an argument of a guard test predicate or a body literal to an argument of a head literal. In the following we give the formal definitions of these primary program dependences.

We assume that the mode information of each argument position in a concurrent logic program has been inferred by the algorithm proposed by Krishna Rao *et al.*, and has the type, *in* or *out* [15]. We use $\mathbf{M}(\alpha)$ to represent the mode of an argument α and $\Gamma(\alpha)$ to represent the set of all variables which appear in an argument α of the program.

3.1.1 Sharing Dependences

Definition 3.1 (Backward-Sharing Dependence) *Let P be a concurrent logic program and u, v be two literals of P such that u is a body literal (or a guard test predicate) and v is a head literal of a clause C of P . Let α and α' be two argument positions such that $\alpha = (C, j, u, k)$ ($j \geq 1$) and $\alpha' = (C, 0, v, k')$. α is backward sharing-dependent on α' iff all of the following conditions hold:*

(1) $\Gamma(\alpha) \cap \Gamma(\alpha') \neq \emptyset$, i.e., there is at least one variable shared by α and α' , and (2) $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = in$.

Example. Considering the clause $C8$ of the program shown in Figure 1. Let $\alpha = (C8, 0, pop, 3)$ represent the third argument “S” of the head literal $pop(N, L, S, OS)$ and $\alpha' = (C8, 3, pop, 2)$ represent the second argument “S” of the second body literal $pop(X, S, NS)$ of the clause, and α and α' are all input argument positions, i.e., $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = in$. α is backward sharing-dependent on α' since there is a shared variable S between α and α' .

Definition 3.2 (Forward-Sharing Dependence) *Let P be a concurrent logic program and u, v be two literals of P such that u is a head literal and v is a body literal (or a guard test predicate). Let α and α' be two argument positions such that $\alpha = (C, 0, u, k)$ and $\alpha' = (C, j, v, k')$ ($j \geq 1$). α is forward sharing-dependent on α' iff all of the following conditions hold: (1) $\Gamma(\alpha) \cap \Gamma(\alpha') \neq \emptyset$, i.e., there is at least one variable shared by α and α' , (2) $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = out$.*

Example. Considering the clause $C8$ of the program shown in Figure 1. Let $\alpha = (C8, 0, pop, 4)$ represent the fourth argument “OS” of the head literal $pop(N, L, S, OS)$ and $\alpha' = (C8, 5, pop, 4)$ represent the fourth argument “OS” of the fourth body literal $pop(NN, NL, NS, OS)$ of the clause, and α and α' are all output argument positions, i.e., $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = out$. α is forward sharing-dependent on α' since there is a shared variable OS between α and α' .

3.1.2 Communication Dependences

Notice that in addition to a shared variable between two arguments in the clause, our definition of a sharing dependence in a single clause also requires that one of the arguments belongs to the head literal and the other belongs to the body literal. As a result, our definition excludes the case that two arguments, which belong to two body literals, share a common variable. We introduce a new type of primary program dependences named *communication dependence* to represent this case. The communication dependence also reflects the data flow in a single clause, and more importantly, reflect the fact of interprocess communications between two body literals.

Definition 3.3 (Communication Dependence) *Let P be a concurrent logic program and u, v be two body literals of a clause C of P . Let α and α' be two argument positions such that $\alpha = (C, j, u, k)$ ($j \geq 1$) and $\alpha' = (C, j', v, k')$ ($j' \geq 1$). α is communication-dependent on α' iff all of the following conditions hold: (1) $\Gamma(\alpha) \cap \Gamma(\alpha') \neq \emptyset$, i.e., there is at least one variable shared by α and α' , (2) $\mathbf{M}(\alpha) = out$ and $\mathbf{M}(\alpha') = in$, and (3) for any argument position α'' of a head literal, there exists no sharing dependence between α'' and α , and also between α'' and α' .*

Intuitively, a communication dependence exists between two arguments belonging to two body literals in a clause that share a common variable such that the variable does not appear in any head literals of the clause.

Example. Considering the clause $C8$ of the program shown in Figure 1. Let $\alpha = (C8, 5, pop, 3)$ represent the third argument “NS” of the fourth body

literal $pop(NN, NL, NS, OS)$ of the clause, and $\alpha' = (C8, 3, pop, 3)$ represent the third argument “NS” of the second body literal $pop(X, S, NS)$, and α is an output argument position, i.e., $\mathbf{M}(\alpha) = out$ and α' is an input argument position, i.e., $\mathbf{M}(\alpha') = in$. α is communication-dependent on α' since there is a shared variable OS between α and α' and the variable NS does not appear in the head literal $pop(N, L, S, OS)$ of the clause.

3.1.3 Unification Dependences

Data as we mentioned above, can be transferred not only in a single clause but also between different clauses via unifications in a concurrent logic programs. The third type of primary program dependence named *unification dependence* are therefore introduced to capture such kind of information and reflect the data flow between arguments in different clauses in a concurrent logic program. Similar to sharing dependences, according to the direction which data is transferred and the mode information of the arguments between two clauses, we also divide the unification dependences into two categories: one which is called *backward-unification dependence* to reflect the data-flow from an input argument position of a head literal of a clause to an input argument position of a body literal of another clause, and the other which is called *forward-unification dependence* to reflect the data-flow from an output argument position of a body literal of a clause to an output argument position of a head literal of another clause.

Definition 3.4 (Backward-Unification Dependence)

Let P be a concurrent logic program and u, v be two literals of clause C of P such that u is a head literal and v is a body literal. Let α and α' be two argument positions such that $\alpha = (C, 0, p, k)$ and $\alpha' = (C', j, p, k)$ ($j \geq 1$). α is backward unification-dependent on α' iff all of the following conditions hold: (1) The unification of u and v does not fail, and (2) $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = in$.

Example. Considering the two clauses $C4$ and $C8$ of the program shown in Figure 1. Let $\alpha = (C8, 0, pop, 3)$ represent the third argument “S” of the head literal $pop(N, L, S, NS)$ of $C8$ and $\alpha' = (C4, 3, pop, 3)$ represent the third argument “S” of the third body literal $pop(N, R, S, OS)$ of $C4$, and α and α' are two input argument positions, i.e., $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = in$. α is backward unification-dependent on α' since there exists a possible unification of the fourth literal $pop(N, R, S, NS)$ of $C4$ and the head literal $pop(N, L, S, OS)$ of $C8$.

Definition 3.5 (Forward-Unification Dependence) *Let P be a concurrent logic program and u, v be two literals of clause C of P such that u is a body literal and v is a head literal. Let α and α' be two argument positions such that $\alpha = (C, j, p, k)$ ($j \geq 1$) and $\alpha' = (C', 0, p, k)$. α is forward unification-dependent on α' iff all of the following conditions hold: (1) The unification of u and v does not fail, and (2) $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = out$.*

Example. Considering the two clauses $C4$ and $C8$ of the program shown in Figure 1. Let $\alpha = (C4, 3, pop, 2)$ represent the second argument “R” of the third body literal $pop(N, R, S, NS)$ of $C4$ and $\alpha' = (C8, 0, pop, 2)$ represent the second argument “L” of the head literal $pop(N, L, S, OS)$ of $C8$, and α and

α' are two output argument positions, i.e., $M(\alpha) = M(\alpha') = out$. α is forward unification-dependent on α' since there exists a possible unification of the fourth literal $pop(N,R,S,NS)$ of $C4$ and the head literal $pop(N,L,S,OS)$ of $C8$.

3.2 The Argument Dependence Net

Program dependence representations such as the *program dependence graph* (PDG) [12] for sequential imperative programs, have many applications in software engineering activities for imperative programs since one can use such representation to explicitly represent various primary program dependences in the programs. In order to use program dependences to develop software engineering tools for concurrent logic programs, here we present a similar representation for concurrent logic programs. The representation is named the argument dependence net which is an arc-classified digraph to represent all primary program dependences in a concurrent logic program. Roughly speaking, the argument dependence net is an extension of the PDG to the case of concurrent logic programs. The net of a concurrent logic program consists of vertices and arcs such that each vertex represents an unique argument position and each arc represents some dependence relationship between arguments in the program.

Definition 3.6 (Argument Dependence Net) *The argument dependence net (ADN) of a concurrent logic program P is an arc-classified digraph (V_A, Sha, Com, Uni) , where V_A is the set of all argument positions of P ; Sha is the set of sharing dependence arcs such that any $(\alpha, \alpha') \in Sha$ iff α is sharing-dependent on α' ; Com is the set of communication dependence arcs such that any $(\alpha, \alpha') \in Com$ iff α is communication-dependent on α' ; Uni is the set of unification dependence arcs such that any $(\alpha, \alpha') \in Uni$ iff α is unification-dependent on α' .*

Example. Figure 2 shows the ADN of the program in Figure 1. Because of the limitation of space, we only show a partial ADN of the program which is related to the fourth clause $C4$ of the program. The rest part of the ADN of the program can be drawn easily and is omitted in the figure. Moreover, we use thin solid arcs to represent sharing dependences, thick dashed arcs to represent communication dependences, and thin dashed arcs to represent unification dependences.

3.3 Building ADNs

In this section we show a concrete algorithm for building an ADN of a concurrent logic program. The algorithm is based on the condition that mode information of the program has been already known. The algorithm consists of three parts, an algorithm *Sha_Com_Dependence* for building a subnet of the ADN concerning sharing and communication dependences in a clause of the program, an algorithm *Uni_Dependence* for building a subnet of the ADN concerning unification dependences between clauses of the program, and an algorithm for building the complete ADN by using algorithms *Sha_Com_Dependence* and *Uni_Dependence*.

We now give the first algorithm *Sha_Com_Dependence* in Fig.3. As input the algorithm gets an output position α , and as

output the algorithm returns a selected input position for each variable v appearing in α .

Then, we give the algorithm *Uni_Dependence* in Fig.3. As input the algorithm gets an input position α . As output the algorithm returns those output positions from which there is an arc to α in the subnet of the ADN.

Finally, we give the algorithm *ADN_Builder* in Fig.4 to build the complete ADN of a concurrent logic program. As input the algorithm gets an input position, and as output the algorithm returns an indicator *ok* to show the success of the construction. *ADN_Builder* calls the algorithms *Sha_Com_Dependence* and *Uni_Dependence* as sub-programs for selecting the set S_1 of local candidate input position for α , and for collecting the whole clause unification information S_2 for all the input positions in S_1 , respectively.

4 Applications

Having ADN as an explicit representation of program dependences in a concurrent logic program, we show some applications based on the ADN of the program. Obviously, some traditional applications which aim at generating efficient code for concurrent logic programs can use the program dependence information analyzed by our approach. Here, in addition to these applications, we point out some new applications all based on the ADN of a concurrent logic program, but handle different problems in a support environment for program analysis of concurrent logic programs. These applications include program slicing, understanding and maintenance, and complexity measurement. Some of these applications are based on the ADN of the program directly, and the others are based on the slices of the program.

4.1 Program Slicing and Understanding

The most direct application of ADN is to slice concurrent logic programs since the explicit representation of various program dependences between arguments in a concurrent logic program makes the ADN well suitable for computing slices of the program.

A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a *slicing criterion*. The parts of a program which have a direct or indirect effect on the values computed at a slicing criterion C are called the *program slice with respect to criterion C*. The process of computing program slices is called *program slicing*.

Weiser [19] first introduced the concept of a program slice. He claims that a slice corresponds to the mental abstractions that people make when they debug a program, and suggests the integration of program slicers in debugging environments. After that, various slightly different notions of program slices and a number of methods to compute slices have been proposed for imperative programs [17] as well as logic programs [9, 16]. In the following, we describe some notions of a static slice of a concurrent logic program. The formal definition of a static slice of a concurrent logic program can be found in [20].

A *reduced literal* of a literal l in a concurrent logic program is a literal l' that is derived from l by replacing zero, or more arguments of l with anonymous variables. A *static slicing criterion* for a concurrent logic program P is a 2-tuple (l, α) , where l is a literal in P and α is

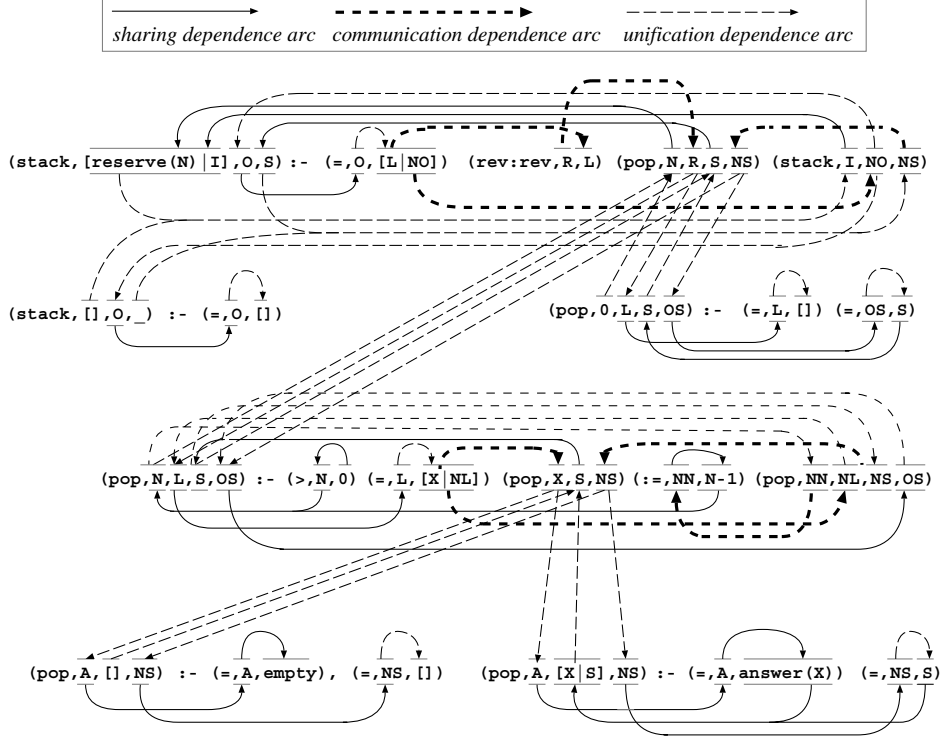


Figure 2: The partial ADN of the program in Figure 1.

an argument position at l . A *static slice* $SS_P(l, \alpha)$ of a concurrent logic program P on a given static slicing criterion (l, α) consists of all reduced literals in P that might affect the values of variables in α at l . Notice that according to the description above a static slice of a concurrent logic program is only a set of reduced literals of the program, and therefore it is an unexecutable slice of the program.

Similarly, we can define a static forward-slicing criterion and a static forward-slice of a concurrent logic program. A *static forward-slice* $SFS_P(l, \alpha)$ of a concurrent logic program P on a given static forward-slicing criterion (l, α) consists of all reduced literals in P that may be affected by the values of variables at α in l . Obviously, a static forward slice of a concurrent logic program is also unnecessarily executable.

Having ADN as a representation of concurrent logic programs, we proposed a program dependence-based approach to slicing concurrent logic program. The main feature of the approach is to define and compute slices of a concurrent logic program by using ADN. Once a concurrent logic program is represented by its corresponding ADN, the slicing of the program can be simplified to a vertex reachability problem in the net and therefore obtained by a linear time forward or backward traverse in the net. Here we briefly describe the slicing algorithm. To compute a static slice or forward-slice, we should refine some notions of a static slice described above based on the ADN of a concurrent logic program.

- Let P be a concurrent logic program, and $N_{ADN} = (V_A, Sha, Com, Uni)$ be the ADN of P . A *static slicing criterion* for N_{ADN} is an argument position $\alpha \in V_A$. The *static slice* $SS_N(\alpha)$ of N_{ADN} on a given

static slicing criterion α is a subset of vertices of V_A , $SS_N(\alpha) \subseteq V_A$, such that for any $\alpha' \in V_A$, $\alpha' \in SS_N(\alpha)$ if and only if there exists a path from α' to α in the ADN.

Similarly, we can define a static forward-slice over the ADN of a concurrent logic program as follows.

- Let P be a concurrent logic program, and $N_A = (V_A, Sha, Uni, Com)$ be the ADN of P . The *static forward-slice* $SFS_N(l, \alpha)$ of N_A on a given static slicing criterion (l, α) is a subset of vertices of V_A , $SFS_N(l, \alpha) \subseteq V_A$, such that for any $\alpha' \in V_A$, $\alpha' \in SFS_N(l, \alpha)$ if and only if there exists a path from α' to α in the ADN.

Notice that in terms of above descriptions, a static backward or forward slice, which is defined over the ADN of a concurrent logic program, is a set of vertices of the ADN. However, since our aim is to obtain a slice of a program, we should map a vertex in the ADN to a reduced literal of the program. Since a vertex in the ADN which represents an argument position of the program has already contained the predicate name of the literal, we can get such mapping quite easily.

- Let P be a concurrent logic program, and $N_{ADN} = (V_A, Sha, Com, Uni)$ be the ADN of P . Let S_N be a static slice over the N_{ADN} , and Σ_L be a set of reduced literals of P , $\beta : S_N \rightarrow \Sigma_L$ is a function from S_N to Σ_L such that for any argument position $\alpha \in S_N$, $\beta(\alpha) \in \Sigma_L$ is a reduced literal of P corresponding to α iff $\beta(\alpha)$ contains α .

```

Procedure Sha_Com_Dependence ( $\alpha, S, ok$ )
  input  $\alpha$ : output position
  output  $S$ : set of pairs (variable  $v$ , input position  $\alpha'$ )
  output  $ok$ : correctness indicator
  begin
     $S := \phi$ ;
     $ok := true$ ;
    Let  $\alpha = (C, j, q, k)$ ;
    for each variable  $v$  in position  $\alpha$  do
       $\alpha' :=$  input position of  $C$  containing  $v$ ;
      if no  $\alpha'$  exists then  $ok := false$ ; return
      else  $S := S \cup (v, \alpha')$  end if;
    end for;
  end

Procedure Uni_Dependence ( $\alpha, S$ )
  input  $\alpha$ : input position
  output  $S$ : set of output positions
  begin
    Let  $\alpha = (C, j, q, k)$ ;
    Let  $l$  be the literal in clause  $C$  that contains  $\alpha$ ;
    if  $ok := 0$  then
      /*  $\alpha$  is an input position of a head literal */
      for each body literal  $l_b$  in a clause  $D$  such that  $l_b$  can unify with  $l$  do
        Let  $\alpha' = (D, j', q, k)$  be the argument position
          corresponding to  $l_{b_k}$  in  $D$ ;
         $S := S \cup \alpha'$ ;
      end for;
    else /*  $\alpha$  is an input position of a body literal */
      for each head literal  $l_h$  in a clause  $D$  such
        that  $l$  can unify with  $l_h$  do
        Let  $\alpha' = (D, 0, q, k)$  be the argument position corresponding to  $l_{h_k}$  in  $D$ ;
         $S := S \cup \alpha'$ ;
      end for;
    end if;
  end

```

Figure 3: The algorithms to compute sharing, communication, and unification dependences.

Having the mapping function β described above, a static slice of a concurrent logic program, which is a set of reduced literals, can be obtained by the mapping function.

Example. Figure 5 shows the partial ADN of the program STACK in Figure 1 and a static slice over the ADN. The slice is represented in shaded vertices in the net and is computed with respect to the slicing criterion $\alpha = (C4, 0, stack, 3)$. Figure 5 shows a static slice of the program STACK in Figure 1 with the slicing criterion $(stack([reverse(N)|I], O, S), S)$ corresponding to the head literal $stack([reverse(N)|I], 0, S)$ of clause C4 of the program in Figure 1. The slice is represented in shaded reduced literals and obtained from the corresponding slice over the ADN in Figure 5 according to the mapping function described above.

When we attempt to understand the behavior of a concurrent logic program, we usually want to know which literals in which clauses might affect a literal of interest, and which literals in which clauses might be affected by the execution of a literal of interest in the program. The slicing and forward-slicing based on the ADN of a concurrent logic program can satisfy these requirements. On the other hand, One of the problems in software maintenance is that of the ripple effect, i.e., whether a code change in a program will affect the behavior of other codes of the program. To maintain a concurrent logic program, it is necessary to know which literals in which clauses will be affected by a modified literal, and which literals in which clauses will affect a modified literal. The needs can be satisfied by slicing

and forward-slicing the program being maintained.

4.2 Complexity Metrics

Software metrics have many applications in software engineering activities including program debugging, testing, analysis, and maintenance, and project management. One could imagine that once some complexity metrics could be proposed for logic programs, they should be helpful in the development of logic programs. Since program dependences are dependence relationships holding between program elements in a program that are determined by control flows and data flows in the program, they can be regarded as one of intrinsic attributes of programs. Therefore it is reasonable to take program dependences as one of objects for measuring program complexity.

In this section, we briefly introduce some complexity metrics which are based on program dependence relations to measure the complexity of a concurrent logic program from various different viewpoints. Detailed definitions can be found in [21]. Once the ADN representation of a concurrent logic program is constructed, the metrics can be computed easily based on the representation. The following notations are used for defining these metrics:

- $Du = Sha \cup Com \cup Uni$.
- $|A|$: the cardinality of set A .
- $\sigma_{[1]=v}(R)$: the selection of binary relation R such that $\sigma_{[1]=v}(R) = \{(v1, v2) | (v1, v2) \in$

```

Procedure ADN_Builder ( $\alpha, ok$ )
  input  $\alpha$ : output position
  output  $ok$ : correctness indicator
begin
  Let  $\alpha = (C, j, q, k)$ ;
  Sha_Com_Dependence( $\alpha, S_1, ok$ );
  if  $ok$  then
    for each pair  $(v, \alpha') \in S_1$  do
      if  $\alpha'$  is already in  $N_{ADN}$ 
      then  $N_{ADN} := N_{ADN} \cup \{(\alpha', \alpha)\}$ 
      else
        repeat
        retry:
          Let  $\alpha' = (C, i, p, l)$ ;
          Push  $\alpha'$  as a vertex into  $N_{ADN}$ ;
          Uni_Dependence( $\alpha', S_2$ );
          for each constructor position  $\alpha \in S_2$  do:
            /*  $\beta$  is functional  $\Rightarrow$  union of nets */
            ADN_Builder( $\beta, ok$ );
            if not  $ok$  then /* backtrack */
              Remove the subnet  $(\alpha' \rightarrow p)$  from  $N_{ADN}$ ;
               $\alpha' :=$  next unexplored input position of  $C$  containing  $v$ ;
              if no  $\alpha'$  exists then return (and fail);
              else exit for-loop end if; /* go to retry */
            end if;
          end for;
        until  $ok$ ;
         $N_{AND} := N_{AND} \cup \{(\alpha', \alpha)\}$ ;
        for each  $\beta = (D, j', q, k) \in S_2$  do
           $N_{AND} := N_{AND} \cup \{(\beta, \alpha')\}$ ;
        end for;
      end if;
    end for;
  end if;
  (else fail)
end if;
end

```

Figure 4: An algorithm to build the complete ADN.

R and $v1 = v\}$.

We first define some metrics for a concurrent logic program that concerns one or more types of program dependences in the program. These metrics can be used to measure various complexities of a concurrent logic program from a general viewpoint.

- $M_1 = |\mathbf{Com}|$: This metric is the number of all primary program dependence concerning interprocess communications in a concurrent logic program. It can be used to measure the complexity of interprocess communications in the program.
- $M_2 = |\mathbf{Uni}|$: This metric is the number of all primary program dependences concerning information flows between clauses via unifications in a concurrent logic program. It can be used to measure the complexity of information flows between clauses in the program.
- $M_3 = |\mathbf{Sha} \cup \mathbf{Com}|$: This metric is the number of all primary program dependences concerning information flows inside clauses due to shared variables in a concurrent logic program. It can be used to measure the complexity of information flows inside clauses of the program.
- $M_4 = |\mathbf{Du}|$: This metric is the number of all primary program dependences in a concurrent logic program. It can be used to measure the total complexity of the program.

Example. The following example shows how to compute the values of the metrics defined above based

on the ADN of a concurrent logic program. To compute these metrics, we need to count the numbers of each type of dependence arcs respectively or the numbers of combination of the dependence arcs in the ADN. The values of the metrics for the sample program in Figure 1 are as follows: $M_1 = |\mathbf{Com}| = 8$. $M_2 = |\mathbf{Uni}| = 24$. $M_3 = |\mathbf{Sha} \cup \mathbf{Com}| = |\mathbf{Sha}| + |\mathbf{Com}| = 23 + 8 = 31$. $M_4 = |\mathbf{Du}| = |\mathbf{Sha} \cup \mathbf{Com} \cup \mathbf{Uni}| = |\mathbf{Sha}| + |\mathbf{Com}| + |\mathbf{Uni}| = 23 + 8 + 24 = 55$.

In maintenance phases, when we have to modify an argument (literal), usually, we intend to know the information about how the modified argument (literal) intersect with other arguments (literals) in the program. This kind of information is very useful because it can tell us if the modified argument (literal) is a special point that connects with its environment more closely than other arguments (literals). If so, that means it is difficult to implement changes to the argument (literal) due to a large number of potential effects on other arguments (literals). We call such an argument (literal) the “most easily affected argument (literal)” of the program. To capture such attribute, we can define the following metric:

- $M_5 = \max\{|\sigma_{[1]=\alpha(\mathbf{D}_u)}| \mid \alpha \in \mathbf{V}_A\}$: This metric is the maximal number of arguments that an argument is somehow dependent on in a concurrent logic program. It can be used to determine the “most easily affected” argument(s) in the program.

Example. The following example shows how to determine the “most easily affected” argument(s) in the program based on its ADN. To do so, we should

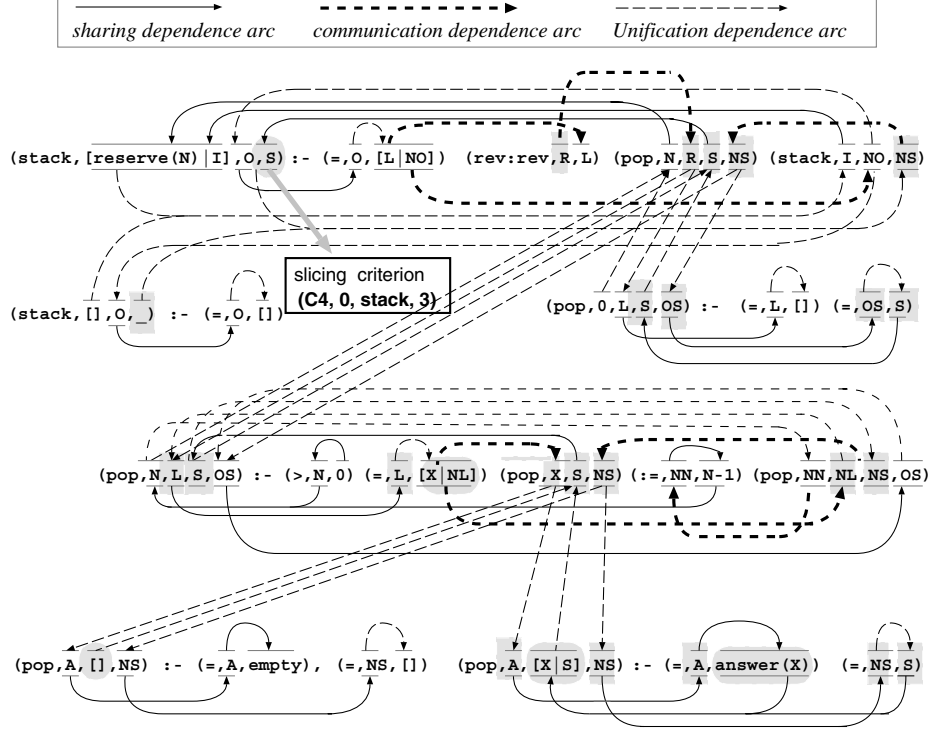


Figure 5: The partial ADN of the program in Figure 1 and a static slice over it.

find those vertices that contains the maximal numbers of dependence arcs in the ADN. By investigating the ADN in Figure 2 of the sample program in Figure 1, we can get $M_5 = \max\{\sigma_{[1]=\alpha}(\mathbf{D}_u) \mid \alpha \in \mathbf{V}_A\} = 3$. $(C8, 0, pop, 1)$, $(C8, 0, pop, 2)$, $(C8, 0, pop, 4)$, $(C8, 2, =, 2)$, $(C8, 3, pop, 1)$, $(C8, 3, pop, 2)$, $(C8, 3, pop, 3)$.

As we observed, all the metrics defined above are absolute metrics. In general, the larger is a metric of a program, the more complex is the program. Moreover, some relative metrics should also be considered since they can measure the complexity of the program from different viewpoint. These new relative metrics can easily be obtained through dividing the above metrics by $|\mathbf{D}_u|$ and $|\mathbf{V}_A|$. Some relative metrics defined by this way are $|\mathbf{Uni}|/|\mathbf{D}_u|$, $|\mathbf{Com}|/|\mathbf{D}_u|$, $|\mathbf{Sha} \cup \mathbf{Com}|/|\mathbf{D}_u|$, and $\max\{\sigma_{[1]=\alpha}(\mathbf{D}_u) \mid \alpha \in \mathbf{V}_A\}/|\mathbf{V}_A|$.

5 CLPKIDS: A Program Analysis System for Concurrent Logic Programs

In this section, we briefly describe the CLPKIDS, a program analysis system that supports development of software engineering tools for concurrent logic programs. The CLPKIDS is a prototype implementation of the techniques described in previous sections. CLPKIDS is written in C and working on UNIX. It consists of five major components at present, that is: a mode analyzer, an ADN builder, a program slicer, a maintenance support tool, and a complexity measurer. In the following, we briefly introduce those tools in CLPKIDS which have been implemented or are being implemented. The detailed implementation issues can be found in [23]

The mode analyzer, *Manalyzer*, is being imple-

mented by using the mode analysis algorithm proposed by Krishna Rao *et al.* [15], and at this time the users have to supply this information via declarations in the program.

The ADN builder, *ADNbuild*, uses mode information outputted by the *Manalyzer* to build the sub-ADN containing only sharing and communication dependence arcs for each clause, and then connects these subnets via unification dependence arcs to form the whole ADN for the program.

The program slicer, *Pslice*, generates various program slices of a source concurrent logic program. At present, it can generate static slices by static slicer *SBslice*, and static forward slices by static forward slicer *SFslice*. Users of the slicer can either access slice information for use in their programs or view the resulting slicer. For a static slice, the user should input a program, a literal and an argument. Since both *SFslice* and *SBslice* require dependence information of the program, they use *ADNbuild*.

The complexity measurer, *Cmeasure*, measures the complexity of a program using various properties of ADN as well as slices. At present this tool provides measured values of complexity of a source program based on the information collected by the *ADNbuild*.

The core modules of the system consists of the program slicer, the maintenance support tool, and the complexity measurer. These modules share some common features: directly (indirectly) depending on program dependence information outputted by the ADN builder *ADNbuild* and also depending on the information of slices produced by the program slicer *Pslice*. These common features allow the analysis performed in an unified framework that simplifies the implementation of

the algorithms. In the next step we intend to extend our system for containing more components such as a graph viewer to display the ADN virtually. We also intend to develop a mechanism to gather the information produced by the mode analyzer, the ADN builder, and the program slicer and put them into an information database where system's users can freely access such information from the database to develop some useful analysis tools for their own purposes.

6 Conclusions

We have proposed a program dependence model for concurrent logic programs. We have presented three types of primary program dependences named the *sharing dependence*, *communication dependence*, and *unification dependence* between arguments in a concurrent logic program and a dependence-based representation named the *argument dependence net* (ADN), which explicitly represents all primary program dependences in a concurrent logic program. We also discussed some applications of the ADN for developing software engineering tools of concurrent logic programs. Finally, we briefly introduced a program analysis system called CLPKIDS for concurrent logic programs, which is a prototype implementation of the techniques introduced in this paper. Although here we presented the program dependences and the representation in terms of KL1, other versions for this approach for more complex concurrent logic languages are easily adaptable because they share their basic execution mechanisms with KL1. The next step for us is to perform some experiments and collect data for evaluating our dependence analysis technique.

References

- [1] Abreu, S. P.: *Improving the Parallel Execution of Logic Programs*, PhD thesis, Universidade Nova de Lisboa (1994).
- [2] Boye, J., Paakki, J. and Maluszynski, J. : Synthesis of Directionality Information for Functional Logic Programs, *Proc. of the Third International Workshop on Static Analysis*, LNCS 724, Springer-Verlag, pp.165-177 (1993).
- [3] Chang, J., Despain, A. M. and Degroot, D. : AND-parallelism of Logic Programs Based on a Static Data Dependency Analysis, *Digest of Papers, COMPCON 85*, IEEE, New York (1985).
- [4] Cheng, J.: Process Dependence Net of Distributed Programs and Its Applications in Development of Distributed Systems, *Proc. IEEE-CS 17th Annual COMP-SAC*, U.S.A., pp.231-240 (1993).
- [5] Debray, S. K. :Static Inference of Modes and Data Dependencies in Logic Programs, *ACM Transaction on Programming Language and System*, Vol.11, No.3, pp.418-450 (1987).
- [6] Debray, S. K. and Lin, N. : Cost Analysis of Logic Programs, *ACM Transaction on Programming Language and System*, Vol.15, No.5, pp.826-875 (1993).
- [7] Ducasse, M. and Noye, J. : Logic Programming Environments: Dynamic Program Analysis and Debugging, *Journal of Logic Programming*, Vol.19/20, pp.351-384 (1994).
- [8] Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The Program Dependence Graph and Its Use in Optimization, *ACM Transaction on Programming Language and System*, Vol.9, No.3, pp.319-349 (1987).
- [9] Gyimothy, T. and Paakki, J. : Static Slicing of Logic Programs, *Local proc. of the 2nd International Workshop on Automated and Algorithmic Debugging*, France (1995).
- [10] King, A. and Soper, P. : Schedule Analysis of Concurrent Logic Programs, *Proc. of the International Joint Conference and Symposium on Logic Programming*, MIT Press, pp.478-492 (1992).
- [11] Korsloot, M. and Tick, E. : Sequentializing Parallel Programs, *Proc. of the Phoenix Seminar and Workshop on Declarative Programming*, Hohrirt, Sasbachwalden, Germany. Springer-Verlag (1991).
- [12] Ottenstein, K. J. and Ottenstein, L. M. : The Program Dependence Graph in a software Development Environment, *ACM Software Engineering Notes*, Vol.9, No.3, pp.177-184 (1984).
- [13] Pereira, L. M. : Rational Debugging in Logic Programming, *Proc. of the 3rd International Logic Programming Conference*, LNCS 225, Springer-Verlag, pp.203-210 (1986).
- [14] Podgurski, A. and Clarke, L. A. : A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance, *IEEE Transaction on Software Engineering*, Vol.16, No.9, pp.965-979 (1990).
- [15] Krishna Rao, M. R. K., Kapur, D. and Shyamasundar, R. K. : Proving Termination of GHC Programs, *Proc. of the Tenth International Conference on Logic Programming*, MIT Press, pp.720-736 (1993).
- [16] Schoenig, S. and Ducasse, M. : A Hybrid Backward Slicing Algorithm Producing Executable Slices for Prolog, *Proc. of the ILPS'95 Workshop on Logic Programming Environments*, Portland, Oregon, USA, December (1995).
- [17] Tip, F. : A Survey of Program Slicing Techniques, *Journal of Programming Languages*, Vol.3, No.3, September, pp.121-189 (1995).
- [18] Ueda, K. and Chikayama, T. :Design of the Kernel Language for the Parallel Inference Machine, *The Computer Journal*, Vol.33, No.6, pp.494-500 (1990).
- [19] Weiser, M.: Program Slicing, *IEEE Transaction on Software Engineering*, Vol.10, No.4, pp.352-357 (1984).
- [20] Zhao, J., Cheng, J. and Ushijima, K. : Slicing Concurrent Logic Programs, in T. Ida, A. Ohori and M. Takeichi (Eds.), *Second Fuji International Workshop on Functional and Logic Programming*, pp.143-162, World Scientific (1997).
- [21] Zhao, J., Cheng, J. and Ushijima, K. : A Metrics Suite for Concurrent Logic Programs, *Proc. 2nd Euro-micro Working Conference on Software Maintenance and Reengineering*, pp.172-178, IEEE Computer Society Press, March 1998.
- [22] Zhao, J.: Slicing Concurrent Java Programs, *Proc. Seventh IEEE International Workshop on Program Comprehension*, pp.126-133, IEEE Computer Society Press, May 1999.
- [23] Zhao, J., Cheng, J. and Ushijima, K. : CLPKIDS: A Program Analysis System for Concurrent Logic Programs, *Proc. 25th IEEE Annual International Computer Software and Applications Conference (COMPSAC'2001)*, IEEE Computer Society Press, October 2001. (to appear)